28 | 接口：接口即契约

2021-12-24 Tony Bai

《Tony Bai · Go语言第一课》 课程介绍 >

讲述：Tony Bai

时长 19:39 大小 18.01M

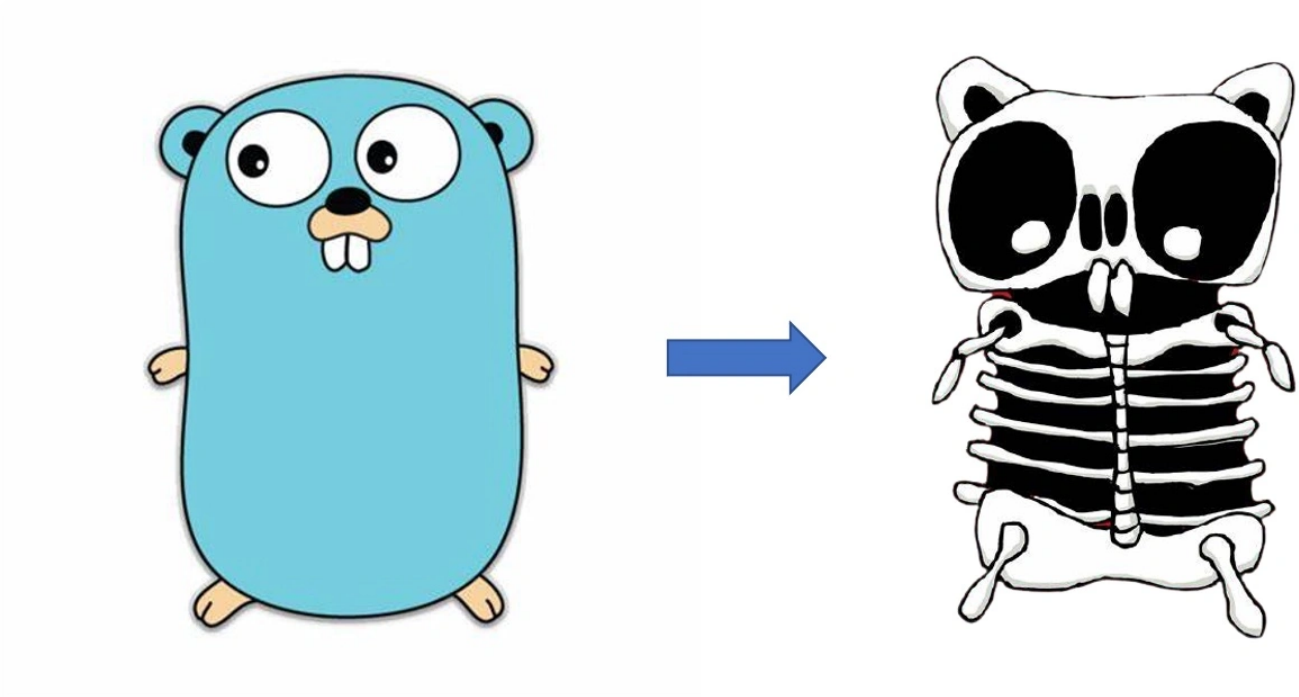
你好，我是 Tony Bai。

从这一讲开始，我们将进入我们这门课核心篇的学习。相对于前两个篇章而言，这篇的内容更加考验大家的理解力，不过只要你跟上节奏，多多思考，掌握核心篇也不是什么困难的事情。

领资料

我先花小小的篇幅介绍一下核心篇的内容。核心篇主要涵盖接口类型语法与 Go 原生提供的三个并发原语（Goroutine、channel 与 select），之所以将它们放在核心语法的位置，是因为它们不仅代表了 Go 语言在编程语言领域的创新，更是**影响 Go 应用骨架（Application Skeleton）设计的重要元素**。

所谓应用骨架，就是指将应用代码中的业务逻辑、算法实现逻辑、错误处理逻辑等“皮肉”逐一揭去后所呈现出的应用结构，这就好比下面这个可爱的 Gopher（地鼠）通过 X 光机所看到的骨骼结构：



通过这幅骨架结构图，你能看到哪些有用的信息呢？从静态角度去看，我们能清晰地看到应用程序的组成部分以及各个部分之间的连接；从动态角度去看，我们能看到这幅骨架上可独立运动的几大机构。

前者我们可以将其理解为 Go 应用内部的耦合设计，而后者我们可以理解为应用的并发设计。而接口类型与 Go 并发语法恰分别是耦合设计与并发设计的主要参与者，因此 Go 应用的骨架设计离不开它们。一个良好的骨架设计又决定了应用的健壮性、灵活性与扩展性，甚至是应用的运行效率。我们后面在讲解接口类型与并发原语的应用模式的时候，还会结合例子深入讲解。

所以，在接下的三讲中，我们将系统学习 Go 语言的接口类型，围绕接口类型的基础知识与接口定义的惯例、接口类型的内部表示以及接口的应用模式这三方面内容进行讲解。在这一讲中，我们先来学习一下接口类型的基础知识部分。

认识接口类型

在前面的学习中，我们曾不止一次接触过接口类型，对接口类型也有了一些粗浅的了解。我们知道，**接口类型是由 type 和 interface 关键字定义的一组方法集合**，其中，方法集合唯一确定了这个接口类型所表示的接口。下面是一个典型的接口类型 MyInterface 的定义：

复制代码

```
type MyInterface interface {
    M1(int) error
    M2(io.Writer, ...string)
}
```

通过这个定义，我们可以看到，接口类型 MyInterface 所表示的接口的方法集合，包含两个方法 M1 和 M2。**之所以称 M1 和 M2 为“方法”，更多是从这个接口的实现者的角度考虑的。**但从上面接口类型声明中各个“方法”的形式上来看，这更像是不带有 func 关键字的函数名 + 函数签名（参数列表 + 返回值列表）的组合。

并且，和我们在21 讲中提到的函数签名一样，我们在接口类型的方法集合中声明的方法，它的参数列表不需要写出形参名字，返回值列表也是如此。也就是说，方法的参数列表中形参名字与返回值列表中的具名返回值，都不作为区分两个方法的凭据。

比如下面的 MyInterface 接口类型的定义与上面的 MyInterface 接口类型定义都是等价的：

复制代码

```
type MyInterface interface {
    M1(a int) error
    M2(w io.Writer, strs ...string)
}

type MyInterface interface {
    M1(n int) error
    M2(w io.Writer, args ...string)
}
```

不过，Go 语言要求接口类型声明中的**方法必须是具名的**，并且**方法名字在这个接口类型的方法集合中是唯一的。**前面我们在学习类型嵌入时就学到过：Go 1.14 版本以后，Go 接口类型允许嵌入的不同接口类型的方法集合存在交集，但前提是交集中的方法不仅名字要一

样，它的函数签名部分也要保持一致，也就是参数列表与返回值列表也要相同，否则 Go 编译器照样会报错。

比如下面示例中 Interface3 嵌入了 Interface1 和 Interface2，但后两者交集中的 M1 方法的函数签名不同，导致了编译出错：

复制代码

```
type Interface1 interface {
    M1()
}
type Interface2 interface {
    M1(string)
    M2()
}

type Interface3 interface{
    Interface1
    Interface2 // 编译器报错：duplicate method M1
    M3()
}
```

看到这里，不知道你有没有注意到，我举的例子中的方法都是首字母大写的导出方法，那在接口类型定义中是否可以声明首字母小写的非导出方法呢？

答案是可以的。**在 Go 接口类型的方法集合中放入首字母小写的非导出方法也是合法的**，并且我们在 Go 标准库中也找到了带有非导出方法的接口类型定义，比如 context 包中的 canceler 接口类型，它的代码如下：

复制代码

```
// $GOROOT/src/context.go

// A canceler is a context type that can be canceled directly. The
// implementations are *cancelCtx and *timerCtx.
type canceler interface {
    cancel(removeFromParent bool, err error)
    Done() <-chan struct{}
}
```

但这样的例子并不多。通过对标准库这为数不多的例子，我们可以看到，如果接口类型的方法集合中包含非导出方法，那么这个接口类型自身通常也是非导出的，它的应用范围也

仅局限于包内。不过，在日常实际编码过程中，我们极少使用这种带有非导出方法的接口类型，我们简单了解一下就可以了。

除了上面这种常规情况，还有空接口类型这种特殊情况。如果一个接口类型定义中没有一个方法，那么它的方法集合就为空，比如下面的 EmptyInterface 接口类型：

复制代码

```
type EmptyInterface interface {

}
```

这个方法集合为空的接口类型就被称为**空接口类型**，但通常我们不需要自己显式定义这类空接口类型，我们直接使用`interface{}`这个类型字面值作为所有空接口类型的代表就可以了。

接口类型一旦被定义后，它就和其他 Go 类型一样可以用于声明变量，比如：

复制代码

```
var err error   // err是一个error接口类型的实例变量
var r io.Reader // r是一个io.Reader接口类型的实例变量
```

这些类型为接口类型的变量被称为**接口类型变量**，如果没有被显式赋予初值，接口类型变量的默认值为 nil。如果要为接口类型变量显式赋予初值，我们就要为接口类型变量选择合法的右值。

Go 规定：**如果一个类型 T 的方法集合是某接口类型 I 的方法集合的等价集合或超集，我们就说类型 T 实现了接口类型 I，那么类型 T 的变量就可以作为合法的右值赋值给接口类型 I 的变量。**在第25和26两讲中，我们已经知道一个类型 T 和其指针类型 *T 的方法集合的求取规则了，所以这里我们也就不难判断一个类型是否实现了某个接口。

如果一个变量的类型是空接口类型，由于空接口类型的方法集合为空，这就意味着任何类型都实现了空接口的方法集合，所以我们可以将任何类型的值作为右值，赋值给空接口类型的变量，比如下面例子：

复制代码

```
var i interface{} = 15 // ok
i = "hello, golang" // ok
type T struct{}
var t T
i = t  // ok
i = &t // ok
```

空接口类型的这一可接受任意类型变量值作为右值的特性，让他成为 Go 加入泛型语法之前**唯一一种具有“泛型”能力的语法元素**，包括 Go 标准库在内的一些通用数据结构与算法的实现，都使用了空类型interface{}作为数据元素的类型，这样我们就无需为每种支持的元素类型单独做一份代码拷贝了。

Go 语言还支持接口类型变量赋值的“逆操作”，也就是通过接口类型变量“还原”它的右值的类型与值信息，这个过程被称为**“类型断言（Type Assertion）”**。类型断言通常使用下面的语法形式：

复制代码

```
v, ok := i.(T)
```

其中 i 是某一个接口类型变量，如果 T 是一个非接口类型且 T 是想要还原的类型，那么这句代码的含义就是**断言存储在接口类型变量 i 中的值的类型为 T**。

如果接口类型变量 i 之前被赋予的值确为 T 类型的值，那么这个语句执行后，左侧“comma, ok”语句中的变量 ok 的值将为 true，变量 v 的类型为 T，它值会是之前变量 i 的右值。如果 i 之前被赋予的值不是 T 类型的值，那么这个语句执行后，变量 ok 的值为 false，变量 v 的类型还是那个要还原的类型，但它的值是类型 T 的零值。

类型断言也支持下面这种语法形式：

复制代码

```
v := i.(T)
```

但在这种形式下，一旦接口变量 i 之前被赋予的值不是 T 类型的值，那么这个语句将抛出 panic。如果变量 i 被赋予的值是 T 类型的值，那么变量 v 的类型为 T，它的值就会是之前

变量 i 的右值。由于可能出现 panic，所以我们并不推荐使用这种类型断言的语法形式。

为了加深你的理解，接下来我们通过一个例子来直观看一下类型断言的语义：

复制代码

```
var a int64 = 13
var i interface{} = a
v1, ok := i.(int64)
fmt.Printf("v1=%d, the type of v1 is %T, ok=%t\n", v1, v1, ok) // v1=13, the t
v2, ok := i.(string)
fmt.Printf("v2=%s, the type of v2 is %T, ok=%t\n", v2, v2, ok) // v2=, the typ
v3 := i.(int64)
fmt.Printf("v3=%d, the type of v3 is %T\n", v3, v3) // v3=13, the type of v3 i
v4 := i.([]int) // panic: interface conversion: interface {} is int64, not []i
fmt.Printf("the type of v4 is %T\n", v4)
```

你可以看到，这个例子的输出结果与我们之前讲解的是一致的。

在这段代码中，如果`v, ok := i.(T)`中的 T 是一个接口类型，那么类型断言的语义就会变成：**断言 i 的值实现了接口类型 T**。如果断言成功，变量 v 的类型为 i 的值的类型，而并非接口类型 T。如果断言失败，v 的类型信息为接口类型 T，它的值为 nil，下面我们再来看一个 T 为接口类型的示例：

复制代码

```
type MyInterface interface {
    M1()
}

type T int

func (T) M1() {
    println("T's M1")
}

func main() {
    var t T
    var i interface{} = t
    v1, ok := i.(MyInterface)
    if !ok {
        panic("the value of i is not MyInterface")
    }
    v1.M1()
    fmt.Printf("the type of v1 is %T\n", v1) // the type of v1 is main.T
```

```
20
21      i = int64(13)
22      v2, ok := i.(MyInterface)
23      fmt.Printf("the type of v2 is %T\n", v2) // the type of v2 is <nil>
24      // v2 = 13 //  cannot use 1 (type int) as type MyInterface in assignment:
25  }
```

我们看到，通过`the type of v2 is <nil>`，我们其实是看不出断言失败后的变量 v2 的类型的，但通过最后一行代码的编译器错误提示，我们能清晰地看到 v2 的类型信息为 MyInterface。

其实，接口类型的类型断言还有一个变种，那就是 **type switch**，不过这个我们已经在第 20 讲讲解 switch 语句的时候讲过了，你可以再去复习一下。

好了，到这里关于接口类型的基础语法我们已经全部讲完了。有了这个基础后，我们再来看看 Go 语言接口定义的惯例，也就是尽量定义“小接口”。

尽量定义“小接口”

接口类型的背后，是通过把类型的行为抽象成**契约**，建立双方共同遵守的约定，这种契约将双方的耦合降到了最低的程度。和生活工作中的契约有繁有简，签署方式多样一样，代码间的契约也有多有少，有大有小，而且达成契约的方式也有所不同。 而 Go 选择了**去繁就简**的形式，这主要体现在以下两点上：

隐式契约，无需签署，自动生效

Go 语言中接口类型与它的实现者之间的关系是隐式的，不需要像其他语言（比如 Java）那样要求实现者显式放置“implements”进行修饰，实现者只需要实现接口方法集合中的全部方法便算是遵守了契约，并立即生效了。

更倾向于“小契约”

这点也不难理解。你想，如果契约太繁杂了就会束缚了手脚，缺少了灵活性，抑制了表现力。所以 Go 选择了使用“小契约”，表现在代码上就是**尽量定义小接口，即方法个数在 1~3 个之间的接口**。Go 语言之父 Rob Pike 曾说过的“接口越大，抽象程度越弱”，这也是 Go 社区倾向定义小接口的另外一种表述。

Go 对小接口的青睐在它的标准库中体现得淋漓尽致，这里我给出了标准库中一些我们日常开发中常用的接口的定义：

复制代码

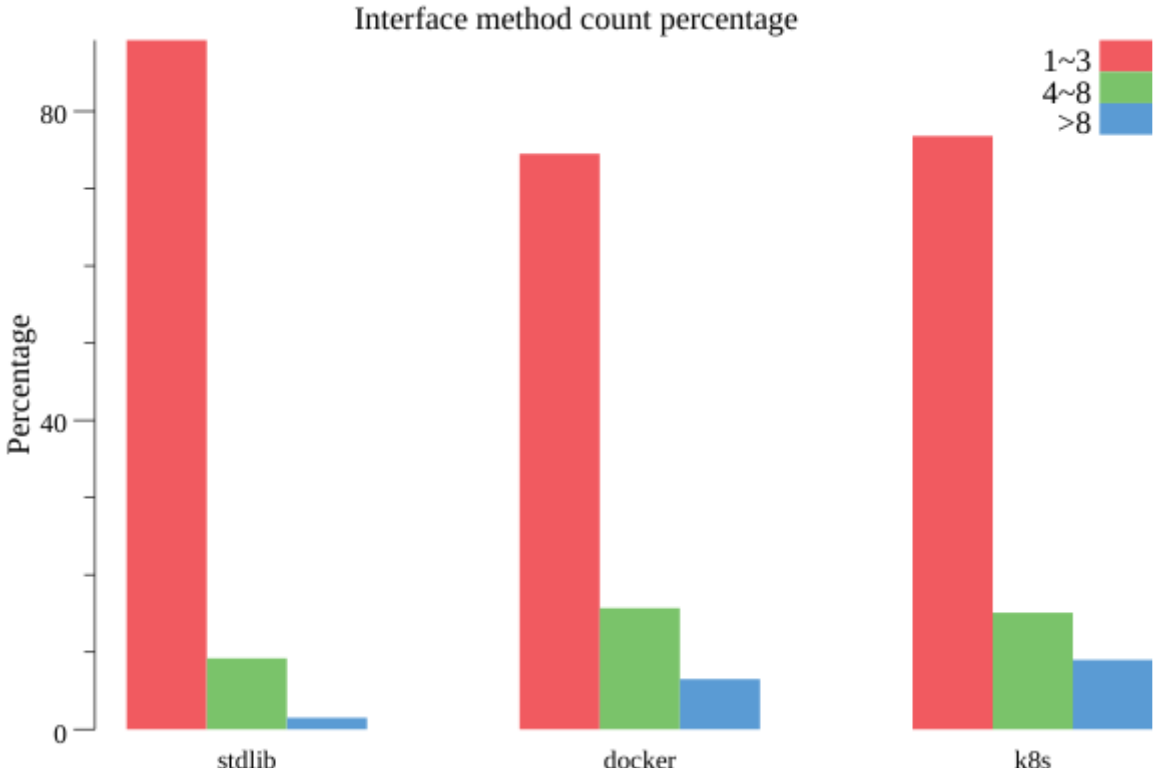
```
// $GOROOT/src/builtin/builtin.go
type error interface {
    Error() string
}

// $GOROOT/src/io/io.go
type Reader interface {
    Read(p []byte) (n int, err error)
}

// $GOROOT/src/net/http/server.go
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}

type ResponseWriter interface {
    Header() Header
    Write([]byte) (int, error)
    WriteHeader(int)
}
```

我们看到，上述这些接口的方法数量在 1~3 个之间，这种“小接口”的 Go 惯例也已经被 Go 社区项目广泛采用。我统计了早期版本的 Go 标准库（Go 1.13 版本）、Docker 项目（Docker 19.03 版本）以及 Kubernetes 项目（Kubernetes 1.17 版本）中定义的接口类型方法集合中方法数量，你可以看下：



从图中我们可以看到，无论是 Go 标准库，还是 Go 社区知名项目，它们基本都遵循了“尽量定义小接口”的惯例，接口方法数量在 1~3 范围内的接口占了绝大多数。

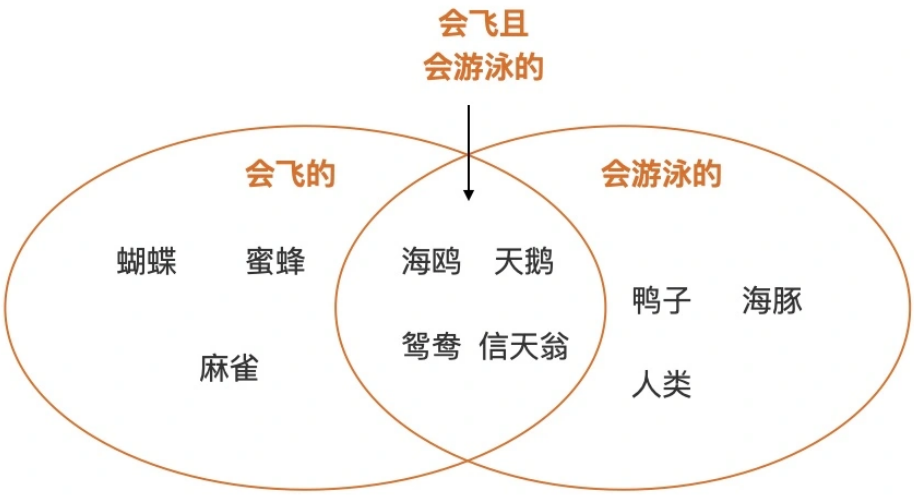
那么在编码层面，小接口究竟有哪些优势呢？这里总结了几点，我们一起来看一下。

小接口有哪些优势？

第一点：接口越小，抽象程度越高

计算机程序本身就是对真实世界的抽象与再建构。抽象就是对同类事物去除它具体的、次要的方面，抽取它相同的、主要的方面。不同的抽象程度，会导致抽象出的概念对应的事物的集合不同。抽象程度越高，对应的集合空间就越大；抽象程度越低，也就是越具像化，更接近事物真实面貌，对应的集合空间越小。

我们举一个生活中的简单例子。你可以看下这张示意图，它是对生活中不同抽象程度的形象诠释：



这张图中我们分别建立了三个抽象：

会飞的。这个抽象对应的事物集合包括：蝴蝶、蜜蜂、麻雀、天鹅、鸳鸯、海鸥和信天翁；

会游泳的。它对应的事物集合包括：鸭子、海豚、人类、天鹅、鸳鸯、海鸥和信天翁；

会飞且会游泳的。这个抽象对应的事物集合包括：天鹅、鸳鸯、海鸥和信天翁。

我们看到，“会飞的”、“会游泳的”这两个抽象对应的事物集合，要大于“会飞且会游泳的”所对应的事物集合空间，也就是说“会飞的”、“会游泳的”这两个抽象程度更高。

我们将上面的抽象转换为 Go 代码看看：

复制代码

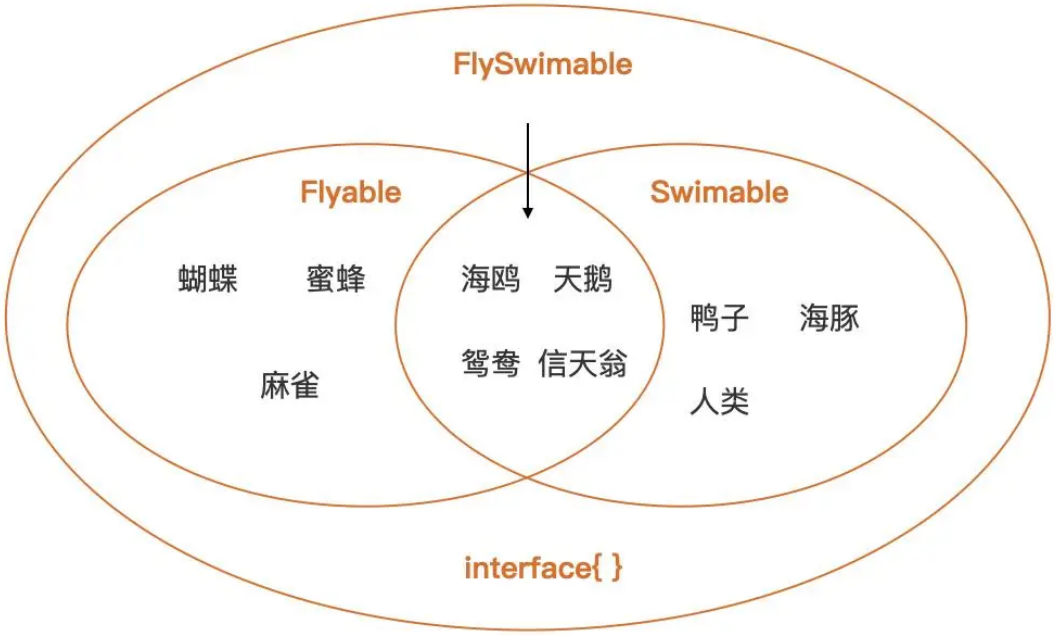
```
// 会飞的
type Flyable interface {
  Fly()
}

// 会游泳的
type Swimable interface {
  Swim()
}

// 会飞且会游泳的
type FlySwimable interface {
```

```
    Flyable
    Swimable
}
```

我们用上述定义的接口替换上图中的抽象，再得到这张示意图：



我们可以直观地看到，这张图中的 Flyable 只有一个 Fly 方法，FlySwimable 则包含两个方法 Fly 和 Swim。我们看到，具有更少方法的 Flyable 的抽象程度相对于 FlySwimable 要高，包含的事物集合（7 种动物）也要比 FlySwimable 的事物集合（4 种动物）大。也就是说，接口越小（接口方法少)，抽象程度越高，对应的事物集合越大。

而这种情况的极限恰恰就是无方法的空接口 interface{}，空接口的这个抽象对应的事物集合空间包含了 Go 语言世界的所有事物。

第二点：小接口易于实现和测试

这是一个显而易见的优点。小接口拥有比较少的方法，一般情况下只有一个方法。所以要想满足这一接口，我们只需要实现一个方法或者少数几个方法就可以了，这显然要比实现拥有较多方法的接口要容易得多。尤其是在单元测试环节，构建类型去实现只有少量方法的接口要比实现拥有较多方法的接口付出的劳动要少许多。

第三点：小接口表示的“契约”职责单一，易于复用组合

我们前面就讲过，Go 推崇通过组合的方式构建程序。Go 开发人员一般会尝试通过嵌入其他已有接口类型的方式来构建新接口类型，就像通过嵌入 io.Reader 和 io.Writer 构建 io.ReadWriter 那样。

那构建时，如果有众多候选接口类型供我们选择，我们会怎么选择呢？

显然，我们会选择那些新接口类型需要的契约职责，同时也要求不要引入我们不需要的契约职责。在这样的情况下，拥有单一或少数方法的小接口便更有可能成为我们的目标，而那些拥有较多方法的大接口，可能会因引入了诸多不需要的契约职责而被放弃。由此可见，小接口更契合 Go 的组合思想，也更容易发挥出组合的威力。

定义小接口，你可以遵循的几点

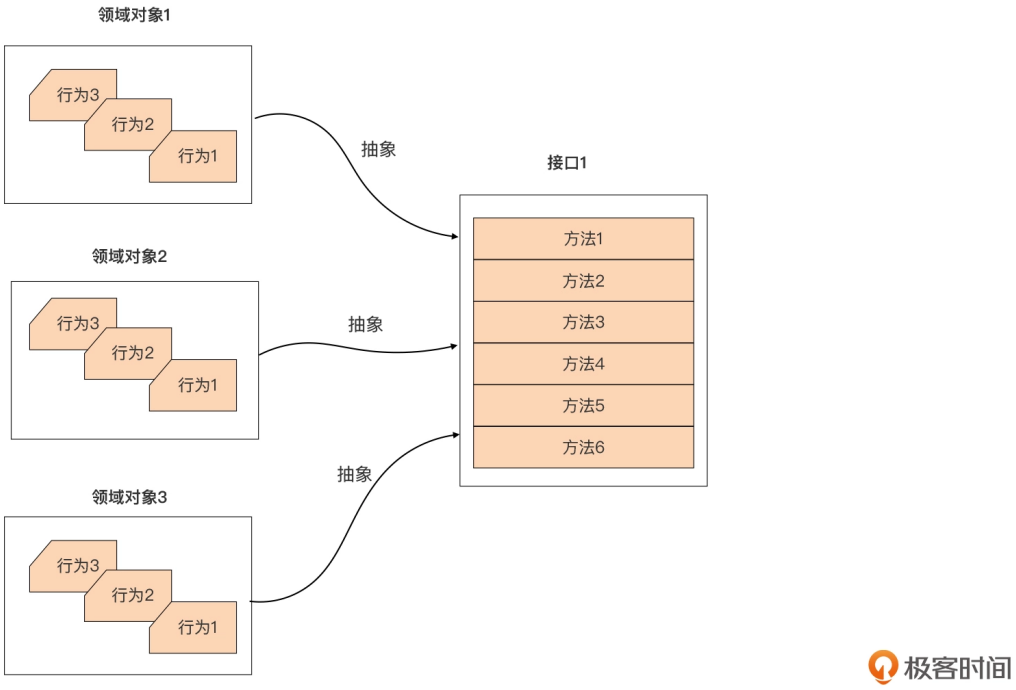
保持简单有时候比复杂更难。小接口虽好，但如何定义出小接口是摆在所有 Gopher 面前的一道难题。这道题没有标准答案，但有一些点可供我们在实践中考量遵循。

首先，别管接口大小，先抽象出接口。

要设计和定义出小接口，前提是需要先有接口。

Go 语言还比较年轻，它的设计哲学和推崇的编程理念可能还没被广大 Gopher 100% 理解、接纳和应用于实践当中，尤其是 Go 所推崇的基于接口的组合思想。

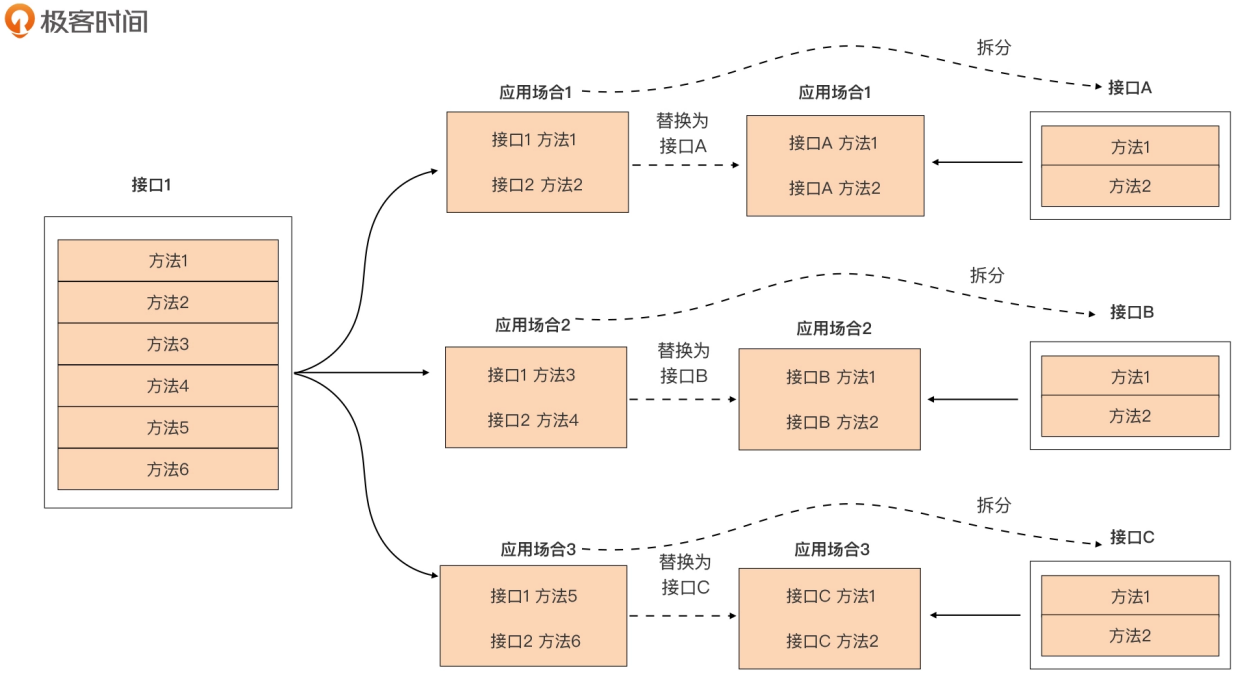
尽管接口不是 Go 独有的，但**专注于接口是编写强大而灵活的 Go 代码的关键**。因此，在定义小接口之前，我们需要先针对**问题领域**进行深入理解，聚焦抽象并发现接口，就像下图所展示的那样，先针对领域对象的行为进行抽象，形成一个接口集合：



初期，我们先不要介意这个接口集合中方法的数量，因为对问题域的理解是循序渐进的，在第一版代码中直接定义出小接口可能并不现实。而且，标准库中的 io.Reader 和 io.Writer 也不是在 Go 刚诞生时就有的，而是在发现对网络、文件、其他字节数据处理的实现十分相似之后才抽象出来的。并且越偏向业务层，抽象难度就越高，这或许也是前面图中 Go 标准库小接口（1~3 个方法）占比略高于 Docker 和 Kubernetes 的原因。

第二，将大接口拆分为小接口。

有了接口后，我们就会看到接口被用在了代码的各个地方。一段时间后，我们就来分析哪些场合使用了接口的哪些方法，是否可以将这些场合使用的接口的方法提取出来，放入一个新的小接口中，就像下面图示中的那样：



这张图中的大接口 1 定义了多个方法，一段时间后，我们发现方法 1 和方法 2 经常用在场合 1 中，方法 3 和方法 4 经常用在场合 2 中，方法 5 和方法 6 经常用在场合 3 中，大接口 1 的方法呈现出一种按业务逻辑自然分组的状态。

这个时候我们可以将这三组方法分别提取出来放入三个小接口中，也就是将大接口 1 拆分为三个小接口 A、B 和 C。拆分后，原应用场合 1~3 使用接口 1 的地方就可以无缝替换为接口 A、B、C 了。

最后，我们要注意接口的单一契约职责。

那么，上面已经被拆分成的小接口是否需要进一步拆分，直至每个接口都只有一个方法呢？这个依然没有标准答案，不过你依然可以考量一下现有小接口是否需要满足单一契约职责，就像 io.Reader 那样。如果需要，就可以进一步拆分，提升抽象程度。

小结

好了，今天的课讲到这里就结束了，现在我们一起来回顾一下吧。

接口类型被列入 Go 核心语法，它的重要性不言自明，它既是 Go 语言的一个微创新，也是影响 Go 应用骨架设计的重要元素。

在这一节课中，我们首先系统学习了接口类型的基础知识，包括接口类型的声明、接口类型变量的定义与初始化以及类型断言等，这里面有很多语法细节，你一定要牢牢掌握，避免后续在使用接口时走弯路。比如，某接口类型定义中嵌入的不同接口类型的方法集合若存在交集，交集中的方法不仅名字要一样，函数签名也要相同。再比如，对接口类型和非接口类型进行类型断言的语义是不完全相同的。

Go 接口背后的本质是一种“契约”，通过契约我们可以将代码双方的耦合降至最低。Go 惯例上推荐尽量定义小接口，一般而言接口方法集合中的方法个数不要超过三个，单一方法的接口更受 Go 社区青睐。

小接口有诸多优点，比如，抽象程度高、易于测试与实现、与组合的设计思想一脉相承、鼓励你编写组合的代码，等等。但要一步到位地定义出小接口不是一件简单的事，尤其是在复杂的业务领域。我这里也给出了循序渐进地抽象出小接口的步骤，你可以参考并在实践中尝试一下。

思考题

1. 如果 Go 中没有接口类型，会发生什么，会对我们的代码设计产生怎样的影响？
2. 关于尽量定义小接口，你有什么好方法？

欢迎在留言区留下你的观点与想法。也欢迎你把这节课分享给更多对 Go 语言的接口感兴趣的朋友。我是 Tony Bai，下节课见。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 2 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 期中测试 | 一起检验下你的学习成果吧

下一篇 孔令飞 | 从小白到“老鸟”：我的Go语言进阶之路

更多学习推荐

2021 最新大厂
Go 工程师面试真题

大厂面试真题 + 金九银十全新整理 + 核心知识全覆盖

免费领取

精选留言 (6)

写留言

罗杰 神

2021-12-24

接口的目的是为了解耦，关于接口在项目还用的非常非常少，几乎没有定义过接口，有点惭愧。

2

AlexWillBeGood

2021-12-24

反射和接口还有即将加入的泛型都可以实现抽象，增加了代码的灵活性

1

return

2021-12-24

抽象就是对同类事物去除它具体的、次要的方面，抽取它相同的、主要的方面。不同的抽象程度，会导致抽象出的概念对应的事物的集合不同。抽象程度越高，对应的集合空间就越大；抽象程度越低，也就是越具像化，更接近事物真实面貌，对应的集合空间越小。

展开

1

功夫熊猫

2021-12-24

嵌套里interface2里不是M1方法的函数列表和interface1的函数列表不一样吗？

共 1 条评论 > 1

lesserror

2021-12-25

请问一下大白老师，一般错误输出的堆栈中都有类似这样的信息：main.go:20 +0x22d，请问像类似 +0x22d 这样的字符 代表的是什么呢？

Calvin

2021-12-25

go1.18 增加了 any 关键字，用以替代现在的 interface{} 空接口类型：type any = interface{}，实际上是 interface{} 的别名。